Corrigé DS3

Partie 1

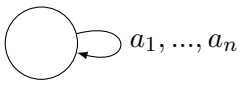
1. Fonction vue et revue en TP.

```c
machine* initialiser_machine(int tQ, int tSigma)
{
    machine* m = (machine*)malloc(sizeof(machine));
    int** delta = (int**)malloc(sizeof(int*)*tQ);
    for (int q = 0; q < tQ; q++)
    {
        delta[q] = (int*)malloc(sizeof(int)*tSigma);
        for (int a = 0; a < tSigma; a++)
        {
            delta[q][a] = q;
        }
    }
    m->taille_Q = tQ;
    m->taille_Sigma = tSigma;
    m->delta = delta;
    return m;
}
```

2. Comme toujours, attention à l'ordre des libérations.

```c
void liberer_machine(machine* m)
{
    for (int q = 0; q < m->taille_Q; q ++)
    {
        free(m->delta[q]);
    }
    free(m->delta);
    free(m);
}
```

3. Une machine est complète donc une machine à un seul état est nécessairement de la forme



où $\Sigma = \{a_1, ..., a_n\}$. Ainsi, tous les mots de $\Sigma^\star$ sont synchronisants dans ce cas particulier.

4. Pour tout $u \in \Sigma^\star$ on montre facilement que :

   - Si $|u| \equiv 0 \mod 2$, alors $\delta^\star(q_i, u) = q_i$ pour $i \in \{0, 1\}$.
   - Si $|u| \equiv 1 \mod 2$, alors $\delta^\star(q_i, u) = q_{1-i}$ pour $i \in \{0, 1\}$

   Ainsi, pour tout mot $u \in \Sigma^\star$, $\delta^\star(q_0, u) \neq \delta^\star(q_1, u)$ donc M_1 n'admet aucun mot synchronisant.

5. Le mot *ada* convient par exemple : il mène systématiquement à q_1.

6. Facile puisque la lecture de `u` dans `m` n'est jamais bloquante. On peut aussi remplacer la boucle pour par une boucle tant que et parcourir le mot `u` jusqu'à rencontrer le symbole de fin de chaîne.

```c
int delta_etoile(machine* m, int q, char* u)
{
    int etat_courant = q;
    int taille_mot = strlen(u);
    for (int i = 0; i < taille_mot; i++)
    {
        int lettre = (int) u[i] -97;
        etat_courant = m->delta[etat_courant][lettre];
    }
    return etat_courant;
}
```

7. Le mot u est synchronisant si et seulement si sa lecture dans chacun des états mène à un seul et même état. On calcule donc l'état q_0 qu'on atteint en lisant u dans l'état 0 (qui existe car une machine a au moins un état) et on vérifie que la lecture de u dans les autres états mène à q_0 aussi :

```c
bool synchronisant(machine* m, char* u)
{
    int etat_candidat = delta_etoile(m,0,u);
    for (int q = 1; q < m->taille_Q; q++)
    {
        if (delta_etoile(m,q,u) != etat_candidat)
        {
            return false;
        }
    }
    return true;
}
```

8. Soit $u \in \Sigma^\star$.

$$u \in S(M) \Leftrightarrow \exists q_0 \in Q, \forall q \in Q, \delta^\star(q,u) = q_0 \Leftrightarrow \exists q_0 \in Q, \overline{\delta}^\star(Q,u) = \{q_0\}$$

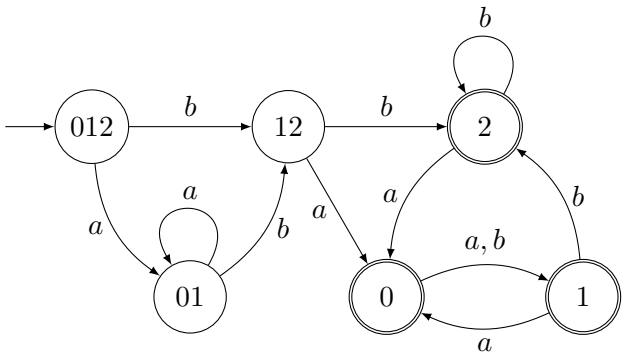
On en déduit qu'il existe un mot dans $S(M)$ si et seulement si il existe un état de $\overline{M}$ étiqueté par un singleton et accessible depuis l'état $Q \in \overline{Q}$ dans la machine $\overline{M}$.

9. On considère l'automate déterministe A dont les états sont donnés par $\overline{Q}$, d'alphabet Σ, dont l'état initial est Q, les états finaux sont donnés par $\{\{q\} \mid q \in Q\}$ et dont la fonction de transition est $\overline{\delta}$. La question précédente montre que A reconnaît $S(M)$; par conséquent ce langage est rationnel en vertu du théorème de Kleene.

10. On procède de manière similaire à celle qu'on utilise pour déterminiser accessiblement. La table suivante est la table des transitions d'un automate reconnaissant $S(M_0)$.

	$\{q_0,q_1,q_2\}$	$\{q_0,q_1\}$	$\{q_1,q_2\}$	$\{q_0\}$	$\{q_1\}$	$\{q_2\}$
a	$\{q_0,q_1\}$	$\{q_0,q_1\}$	$\{q_0\}$	$\{q_1\}$	$\{q_0\}$	$\{q_0\}$
b	$\{q_1,q_2\}$	$\{q_1,q_2\}$	$\{q_2\}$	$\{q_1\}$	$\{q_2\}$	$\{q_2\}$

On obtient ainsi l'automate déterministe (non minimal) suivant, pour lequel on vérifie rapidement que ba et bb sont reconnus afin de détecter une éventuelle erreur d'inattention.



Partie 2

11. Dans les deux cas, on se sert de la taille de la file pour en déduire son état :

```
let est_vide (f:'a file) :bool = f.taille = 0
```

```
let est_pleine (f:'a file) :bool = f.taille = Array.length f.donnees
```

12. Si la file n'est pas pleine, on place l'élément x à la première position circulairement libre puis on modifie la valeur de taille pour prendre en compte le fait qu'il y a un élément de plus dans la file.

```
let enfiler (f:'a file) (x:'a) :unit =
    if est_pleine f then failwith "La file est pleine"
    else begin
        let n = Array.length f.donnees in
        let i = (f.debut + f.taille) mod n in
        f.donnees.(i) <- x;
        f.taille <- f.taille +1;
    end
```

13. Si la file n'est pas vide, on récupère sa tête puis on modifie les valeurs de debut et taille.

```
let defiler (f:'a file) :'a =
    if est_vide f then failwith "La file est vide"
    else begin
        let n = Array.length f.donnees in
        let tete = f.donnees.(f.debut) in
        f.debut <- (f.debut + 1) mod n;
        f.taille <- f.taille -1;
        tete
    end;
```

14. Toutes les opérations précédentes se font en temps constant.

Partie 3

15. Le seul point à montrer est la terminaison de la boucle tant que en ligne 12. Cette dernière provient du fait qu'un élément de $s \in S$ est ajouté au plus une fois à la file F : soit lors de la boucle pour de la ligne 7, soit lors de la boucle tant que si $D[s]$ vaut ∞, mais par la suite $D[s]$ ne vaut plus jamais ∞ d'après la ligne 15. Plus formellement, on montre que $(\{|s \in S \,|\, D[s] = \infty\}, |F|)$ diminue strictement pour l'ordre lexicographique au cours de la boucle tant que.

16. La boucle pour de la ligne 7 nécessite $O(|S|)$ opérations. Si on implémente les files via la partie 2, la question 14 assure que la ligne 13 s'effectue en $O(1)$ et il en va de même pour chaque itération de la boucle pour de la ligne 14. Ainsi, le coût du traitement d'un sommet s dans la file F lors d'une itération de la boucle tant que est majoré par un $O(\deg(s))$. Comme chacun des sommets passe au plus une fois dans la file F, le coût total de la boucle tant que est de l'ordre de

$$\sum_{s\in S} \deg(s) = O(|A|)$$

On obtient donc une complexité en $O(|S| + |A|)$ ce qui n'a rien d'étonnant puisque l'algorithme mystère n'est rien d'autre qu'un parcours en largeur depuis l'ensemble X.

17. Prouvons ce résultat par récurrence sur le nombre de passage dans ladite boucle. Avant le premier passage dans la boucle c'est acquis : la file contient tous les éléments $s \in X$ et pour chacun $D[s] = 0$.

Si la propriété est vraie avant le k-ème passage, montrons qu'elle reste vraie après le $(k+1)$-ème (sous réserve qu'il existe). Avant le k-ème passage, la file contenait les éléments $v_1, ..., v_r$. Etant donné le corps de la boucle tant que, avant le $(k+1)$-ème passage, la file contient à présent les éléments $v_2, ..., v_r, w_1, ...w_s$ avec pour tout $i \in [\![1, s]\!]$, $D[w_i] = D[v_1] + 1$.

Les inégalités $D[v_2] \leq ... \leq D[v_r]$ restent vraies par hypothèse de récurrence. Toujours par hypothèse, $D[v_r] \leq D[v_1] + 1 = D[w_1] = ... = D[w_s]$. Donc la chaîne d'inégalités demandée au rang $k+1$ est avérée. Par ailleurs, si $r \geq 2$ alors $D[w_s] - D[v_2] = D[v_1] + 1 - D[v_2] \leq 1$ puisque $D[v_1] \leq D[v_2]$. Sinon, on a bien $D[w_s] - D[w_1] = 0 \leq 1$. L'invariant est donc bien conservé.

18. Procédons par double implication. Montrons d'abord par récurrence sur $D[s]$ que, si $D[s] \neq \infty$ en fin d'algorithme, alors s est accessible depuis X et $d_s \leq D[s]$.

Si $D[s] = 0$ alors $S \in X$ car un sommet $v \notin X$ vérifie $D[v] \geq 1$ par récurrence, l'hérédité étant assurée par la ligne 15. Dans ce cas le résultat est prouvé puisque $d_s = 0$. Supposons le résultat vrai pour tout sommet s' tel que $D[s'] = k$ et considérons un sommet s tel que $D[s] = k+1$. L'algorithme nous informe alors que s admet nécessairement un voisin s' tel que $D[s'] = k$ (ligne 15). En particulier, par hypothèse de récurrence, s est accessible depuis X puisque on peut prolonger le chemin de X à s' en un chemin C de X à s via l'arc liant s' à s. De plus,

$$d_s \leq \text{longueur de } C \leq d_{s'} + 1 \leq D[s'] + 1 = D[s]$$

la dernière égalité résultant de la ligne 15 et la dernière inégalité de l'hypothèse de récurrence.

Réciproquement, montrons par récurrence sur d_s que si s est accessible depuis X alors $D[s] \neq \infty$. Si $d_s = 0$ alors $s \in X$ et la boucle pour de la ligne 7 assure que $D[s] = 0 \neq \infty$. Supposons que le résultat est avéré pour tout sommet s' tel que $d_{s'} = k$ et considérons un sommet s tel que $d_s = k+1$. Alors il existe un chemin allant d'un élément $x \in X$ au sommet s de longueur $k+1$ et ce chemin se décompose en un chemin C de x à un certain sommet s' voisin de s complété par l'arc (s', s). Le chemin C est un sous-chemin d'un plus court-chemin donc est un plus court chemin ; ainsi, $d_{s'} = k$ et on peut utiliser l'hypothèse de récurrence pour affirmer que $D[s'] \neq \infty$ donc que s' passe dans la file F. Lorsque s' est défilé :

- Soit $D[s]$ est déjà différent de ∞ à ce moment.
- Soit $D[s] = \infty$ à ce moment mais dans ce cas la valeur de $D[s]$ est modifiée en $D[s'] + 1 \neq \infty$.

Dans tous les cas, $D[s]$ est bien différent de ∞ en fin d'algorithme.

19. Si s est inacessible depuis X, alors la question 19 montre qu'en fin d'algorithme, $D[s] = \infty = d_s$.

Supposons par l'absurde qu'il existe un sommet v accessible depuis X tel que $D[v] \neq d_v$, c'est-à-dire tel que $d_v > D[v]$ d'après la question 19. Considérons dès lors s, le premier sommet vérifiant cette propriété à être mis dans la file F au cours de l'algorithme.

Nécessairement, $s \notin X$, sinon on aurait $D[s] = 0 = d_s$. On en déduit que s a été rajouté à la file lors du traitement d'un sommet s' qui a donc été placé dans la file avant s. Par construction de s, $D[s'] = d_{s'}$. Le fonctionnement de l'algorithme assure en outre qu'il existe un chemin de X à s passant par s' et en particulier un chemin de taille $d_{s'} + 1$ de X à s donc

$$d_s \leq d_{s'} + 1 = D[s'] + 1 = D[s] < d_s$$

la dernière égalité provenant de l'exécution de la ligne 15 lors du traitement de s' et la dernière inégalité de l'hypothèse, qu'on vient par là même de montrer absurde.

20. La valeur de c correspond au nombre de sommets non accessibles depuis X ; en effet, c est initialisé avec $|S|$ et décroît de un à chaque fois qu'un sommet jusque là non accessible est atteint.

    Le contenu de $P[s]$ représente l'arête qui précède s dans un plus court chemin de X à s.

21. On traduit en Ocaml le pseudo-code fourni par l'énoncé.

```
let accessibles (g:graphe) (x:int list) =
    let n = Array.length g in
    let f = {donnees = Array.make n 0; debut = 0; taille = 0}
    and d = Array.make n (-1)
    and p = Array.make n (-2,-1)
    and c = ref n in
    let ajouter_sommet s =
        enfiler f s;
        d.(s) <- 0;
        p.(s) <- (-1,-1);
        c := !c-1;
    in List.iter ajouter_sommet x;
    let traiter_arete s (t,a) =
        if d.(t) = -1 then
        begin
            d.(t) <- d.(s) + 1;
            p.(t) <- (s,a);
            enfiler f t;
            c := !c-1;
        end
    in
    while not (est_vide f) do
        let s = defiler f in
        List.iter (traiter_arete s) g.(s)
    done;
    !c, d, p
```

Quelques commentaires :

- On a justifié en question 15 que chaque sommet de G passe au plus une fois dans la file F ; la taille de cette file sera donc toujours inférieure à $|S|$ d'où le choix pour l'initialisation de la taille du champ `donnees` de la file `f`.
- La fonction `ajouter_sommet : int -> unit` permet d'ajouter un sommet de X à la file F selon les modalités de la boucle pour de la ligne 7.
- La fonction `traiter_arete : int -> (int*int) -> unit` correspond au traitement d'une arête lors d'une itération de la boucle pour de la ligne 14.

22. La reconstruction du mot demandé à partir du tableau P se fait depuis la fin du mot : on utilise une fonction auxiliaire construit_mot prenant en argument le mot u en cours de construction et le sommet v tel que la lecture de u dans v amène au sommet s tout en étant sur un plus court chemin de X à s. L'argument u fait office d'accumulateur pour cette fonction.

```
let chemin (s:int) (p:(int*int) array) :int list =
    let rec construit_mot (u:int list) (v:int) = match p.(v) with
        |(-2,-1) -> failwith "sommet non accessible"
        |(-1,-1) -> u
        |(v',lettre) -> construit_mot (lettre::u) v' in
    construit_mot [] s
```

Partie 4

23. On a nécessairement $|Q| = |\delta^\star(Q, u_0)| \geq |\delta^\star(Q, u_1)| \geq ... \geq |\delta^\star(Q, u_r)| = 1$. En effet, la machine étant déterministe, en lisant un mot depuis un état on ne peut atteindre qu'un seul état. Ainsi, en lisant un mot depuis un ensemble d'états, le nombre d'états atteignables ne peut que diminuer au fil de la lecture (en cas de collision). Enfin, comme $u = u_r$ est synchronisant, $|\delta^\star(Q, u_r)| = 1$ par définition.

24. Pour le sens direct, on prend pour tous $q, q' \in Q$, $u_{q,q'} = u$, qui convient pusique u est synchronisant.

    Réciproquement, considérons les suites $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ de mots et la suite $(Q_n)_{n\in\mathbb{N}}$ d'ensembles d'états de M définies récursivement comme suit :

    - $u_0 = v_0 = \varepsilon$ et $Q_0 = Q$.
    - Si $i \in \mathbb{N}$ vérifie $|Q_i| = 1$, on arête le processus de construction. Sinon, il existe deux états différents $q, q' \in Q_i$ et par hypothèse un mot $u_{q,q'}$ vérifiant $\delta^\star(q, u_{q,q'}) = \delta^\star(q', u_{q,q'})$. On pose $u_{i+1} = u_{q,q'}$, $v_{i+1} = v_i u_{i+1}$ et $Q_{i+1} = \delta^\star(Q_i, u_{i+1})$.

    La suite des $(|Q_i|)$ est strictement décroissante et minorée par 1. Le premier point provient du fait que $|Q_{i+1}| = |\delta^\star(Q_i, u_{i+1})| < |Q_i|$ puisque le mot u_{i+1} amène au même état qu'il soit lu dans $q \in Q_i$ ou dans $q' \in Q_i$ et ces deux états sont différents par construction.

    On en déduit qu'à partir d'un certain rang $j < n$, $|Q_j| = 1$. Alors le mot v_j est synchronisant pour la machine M car $\delta^\star(Q, v_j) = \delta^\star(Q_0, v_j) = Q_j$ par définition des trois suites considérées. Ainsi, lire v_j depuis n'importe quel état amène au seul état de Q_j.

25. Il y a $\binom{n}{1} = n$ parties à un élément de Q et $\binom{n}{2} = n(n-1)/2$ parties à 2 éléments de Q. L'entier $\widetilde{n}$ est la somme de ces deux quantités :
$$\widetilde{n} = \frac{n(n+1)}{2}$$

26. On peut proposer par exemple la fonction suivante (ce n'est pas la plus simple !) :
$$\varphi_n : \begin{cases} \widetilde{Q} & \longrightarrow [\![0, \widetilde{n}-1]\!] \\ \{q\} & \longmapsto \frac{q(q+1)}{2} \\ \{q_1, q_2\} & \longmapsto \frac{q_1(q_1+1)}{2} + q_2 + 1 \text{ si } q_2 < q_1 \end{cases}$$

    Sa bijectivité est assurée par le fait que $\{[\![\frac{q(q+1)}{2}, \frac{q(q+1)}{2} + q]\!] \,|\, q \in [\![0, n-1]\!]\}$ forme une partition de $[\![0, \widetilde{n}-1]\!]$. On en déduit illico le code suivant :

```
let rec phi (n:int) (l:int list) :int = match l with
    |[q] -> (q * (q+1))/2
    |[q1;q2] when q1 > q2 -> (phi n [q1]) + q2 + 1
    |[q1;q2] when q1 = q2 -> phi n [q1]
    |[q1;q2] -> phi n [q2;q1]
    |_ -> failwith "ceci n'est pas un état de Q tilde"
```

Le troisième cas est a priori inutile mais est ajouté pour faciliter la conception de la fonction suivante.

27. On distingue les cas selon que X est une paire ou un singleton. Si X est une paire telle que la lecture de la lettre a depuis chacun de ses deux états donne le même état, on n'a pas besoin de faire de cas particulier car il est pris en compte par le troisième cas de phi.

```
let delta_tilde (m:machine) (q:int) (a:int) :int =
    let n = Array.length m in
    match (phi_inv n q) with
        |[x] -> phi n [m.(x).(a)]
        |[x;y] -> phi n [m.(x).(a); m.(y).(a)]
```

28. La taille du tableau représentant la machine m donne son nombre d'états n, à partir duquel on calcule $\widetilde{n}$ grâce à la question 25. On créé ensuite les listes d'adjacence décrivant G^T en parcourant tous les états de $\widetilde{M}$ et toutes les lettres et en renversant les arcs fournis par delta_tilde.

```
let transpose (m:machine) :graphe =
    let n = Array.length m and nb_lettres = Array.length m.(0) in
    let n_tilde = n * (n+1) /2 in
    let gt = Array.make n_tilde [] in
    for q = 0 to n_tilde -1 do
        for a = 0 to nb_lettres -1 do
            let q' = delta_tilde m q a in
            gt.(q') <- (q,a)::gt.(q')
        done
    done;
    gt
```

29. Le résultat de la question 24 se reformule de la façon suivante : M admet un mot synchronisant si et seulement si pour tout état X de $\widetilde{M}$ qui est une paire, il existe une transition dans $\widetilde{M}$ permettant de passer de X à un état de $\widetilde{M}$ correspondant à un singleton. Cette dernière assertion est équivalente à dire que, dans $\widetilde{G}^T$, tous les états sont accessibles depuis les singletons.

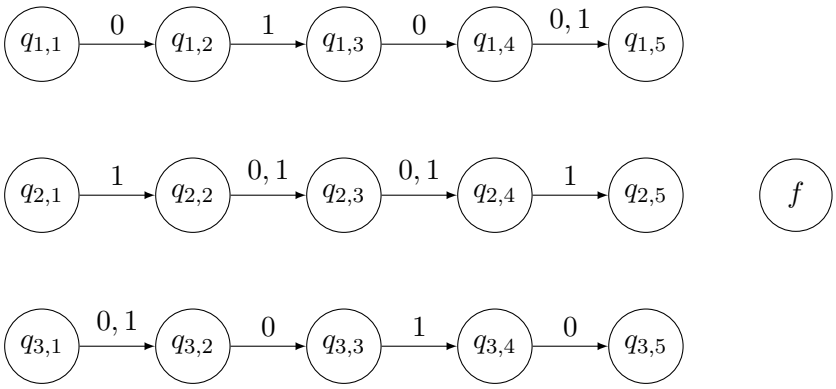
    On a vu en question 20 que la fonction accessibles calcule entre autres un entier c correspondant au nombre d'états non accessibles depuis l'ensemble X considéré dans le graphe d'automates G. On a ainsi l'équivalence suivante : M admet un mot synchronisant si et seulement si appliquer accessibles à $\widetilde{G}^T$ et l'ensemble d'états de $\widetilde{G}^T$ correspondant aux singletons dans M produit un entier c nul.

30. La correction de la fonction suivante provient de la question 29. La fonction auxiliaire singletons permet de créer la liste des sommets de $\widetilde{G}^T$ correspondant à des singletons de Q où Q est l'ensemble d'états de M : on utilise pour ce faire la fonction de traduction phi de la question 26.

```
let existe_synchronisant (m:machine) :bool =
    let gt = transpose m and n = Array.length m in
    let rec singletons (k:int) :int list = match k with
        |0 -> []
        |k -> (phi n [k-1])::(singletons (k-1))
    in let c,_,_ = accessibles ft (singletons n) in
    c = 0
```

Partie 5

31. Toutes les transitions qui ne sont pas dessinées mènent à l'état f.



32. La valuation $v : (x_1, x_2, x_3, x_4) \mapsto (1, 1, 0, 0)$ satisfait φ_0. La lecture du mot 1100 dans M_{φ_0} amène dans l'état puits quel que soit l'état de départ : il est donc synchronisant pour cette machine.

33. Soit u un mot de longueur $m+1$. Montrons que u mène à l'état f à partir de tout état de M_φ. La lecture d'une lettre depuis $q_{i,j}$ mène soit directement à f (et dans ce cas on y reste puisque c'est un puits) soit à l'état $q_{i,j+1}$. S'il existait $(i, j) \in [\![1, n]\!] \times [\![1, m+1]\!]$ tel que lire u depuis $q_{i,j}$ ne mène pas à f, alors cette lecture mènerait à $q_{i,j+m+1}$. Mais $j \geq 1$ donc $j + m + 1 \geq m + 2$ et donc l'état précédent n'existe pas. Par conséquent, tout mot de longueur $m+1$ est synchronisant pour M_φ.

    Si u est un mot de longueur m, le même raisonnement montre que la lecture de u depuis $q_{i,j}$ avec $j \geq 2$ mène à f. Le mot u est donc synchronisant si et seulement si pour tout $i \in [\![1, n]\!]$, $\delta^\star(q_{i,1}, u) = f$.

34. Supposons φ satisfiable via la valuation $(v_1, ..., v_m) \in \{0, 1\}^m$. Montrons que $v = v_1...v_m$ (qui est bien de longueur m) est synchronisant pour M_φ. D'après la question précédente, il suffit de montrer pour ce faire que pour tout $i \in [\![1, n]\!]$, $\delta^\star(q_{i,1}, v) = f$.

    Soit donc $i \in [\![1, n]\!]$. La clause C_i est satisfaite donc d'après la table de vérité du connecteur $\vee$, il existe $j \in [\![1, m]\!]$ tel que ($v_j = 1$ et x_j intervient dans C_i) ou ($v_j = 0$ et $\overline{x_i}$ intervient dans C_i). Considérons le premier tel j. Alors :

$$\delta^\star(q_{i,1}, v) = \delta^\star(\delta^\star(q_{i,1}, v_1...v_{j-1}), v_j...v_m) = \delta^\star(q_{i,j}, v_j...v_m) = f$$

    l'avant-dernière égalité provenant de la minimalité de j et la dernière de la définition de δ et de j.

35. Supposons qu'on dispose d'un mot synchronisant $v = v_1...v_k$ avec $k \leq m$ pour M_φ. Comme la lecture de v dans l'état f mène à f (c'est un puits), l'état commun vers lequel la lecture de v mène est nécessairement f. En particulier, pour tout $i \in [\![1, n]\!]$, $\delta^\star(q_{i,1}, v) = f$. Montrons que la valuation attribuant comme valeur de vérité v_i à la variable x_i si $i \leq k$ et une valeur quelconque sinon satisfait φ.

Supposons par l'absurde que ce ne soit pas le cas. Alors il existe i tel que la clause C_i ne soit pas vérifiée. La définition de δ implique alors que $\delta^\star(q_{i,1}, v) = q_{i,k+1} \neq f$ ce qui est une contradiction.

Remarque : Cette partie montre que le problème "Mot synchronisant" est au moins aussi dur que CNF-SAT. Comme ce dernier est NP-complet, le problème "Mot synchronisant" est donc lui aussi un problème difficile.